Side Scrolling UFO Game Using NeuroEvolution of Augmenting Topologies (NEAT) in Python

Kushagra Dixit (qq328511)

Rahim Jiwa (qq328596)

SCS 3547-05

Intelligent Agents and Reinforcement Learning

Introduction

Side scrolling games based on object avoidance and human reflexes have been a staple of computer games ever since the advent of commercialized video games in the 1980's. Side-scrollers for a lot of games allow for infinite game-play as the obstruction for the player can be continuously created. This allows for the human player to create an infinitely high score until the player makes a single mistake and loses the score streak. As seen in the figure below. The dinosaur in the dinosaur game that loads up when google chrome goes offline has a similar structure. The dinosaur keeps scrolling to the right only to encounter obstacles like the cacti. To keep up, the dinosaur has to jump at precise moments as the game speeds up. This propels the player to make the jump action ahead of time to have a bigger jump arc, thereby increasing the difficulty of the game. Potentially, an Artificially Intelligent algorithm, once fine-tuned, can play the game indefinitely.

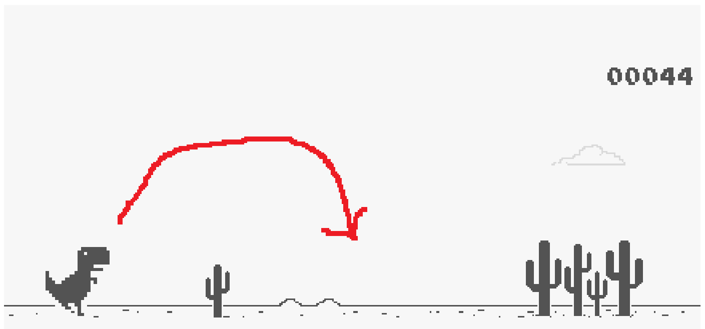


Figure 1.0: Google Chrome Dinosaur game

Genetic algorithms allow for a population of candidate solutions to help optimize the solution set that will allow for infinite gameplay. A population of candidates is used with different phenotypes to solve an optimization problem. The evolutionary selection criteria generally starts from a population of randomly generated individuals and is iterative in nature. The population in each iteration is called a generation and fitness is evaluated for each generation. Fitness can also be labelled as the objective function in the optimization problem being solved. The "more fit" candidates/players are selected from the current generation and each individual's current genome is modified (randomly or by recombination). These mutations allow for variations in each iteration that will allow for better fitness across generations. In the context of a side scrolling game, a genetic algorithm will start with a 100 players, out of which 95% die as they are killed by the first few oncoming obstacles, this will be followed by only 1 or two players that will move to the next obstacle. When all the players die due to the first few obstacles, the hyper-parameters of the fittest candidates are copied. The subsequent generation then allows for some mutations where the entire population performs better thereby increasing the overall fitness score. This gradual improvement in fitness scores leads to optimization

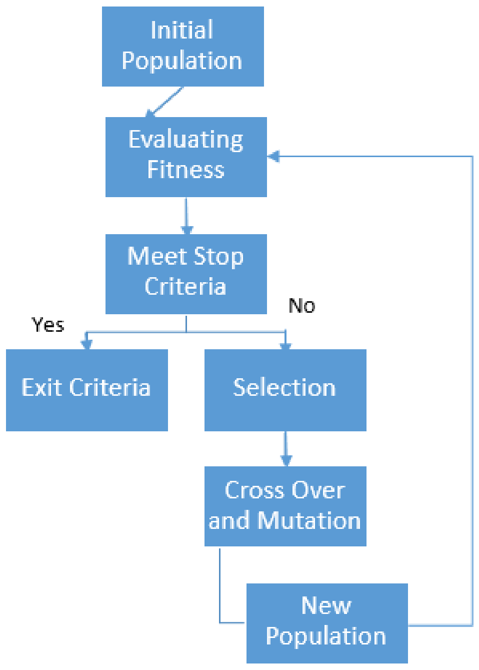
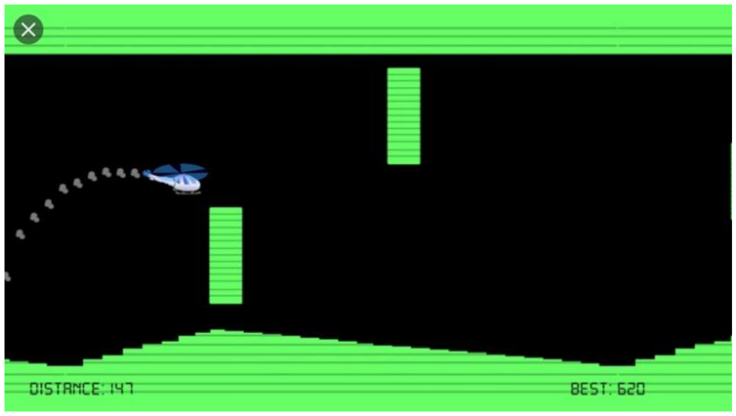


Figure 1.1: Flow chart of Genetic algorithm for optimization

of the allowing for some variations



A good example of an infinite side scrolling game with object avoidance is the helicopter game, where the helicopter adjusts its position on the screen to avoid the green obstacles and terrain. The score can be measured in distance the helicopter covers. Each click by the user initiates a jump arc where the helicopter slightly moves up. if left untouched, the helicopter falls down due to the effects of gravity. Hence, the user needs to calculate the precise timing to press the mouse button. The player intuitively presses the button with precise timing to develop a "feel" as to what would be the best way to avoid an obstacle. As the game progresses, the random appearance of the terrain and obstacles can use the player to complete very hard frame-perfect presses of the jump button. The game never ends and so the user can only compete for the highest score as measured by the distance covered without crashing.

To create a perfect player using artificially intelligent algorithms, NEAT (Neuro-Evolution of Augmenting Topologies) was utilized. It is an evolutionary algorithm that creates artificial neural networks. [1] Evolving artificial neural networks with genetic algorithms also called "NeuroEvolution" has been shown to be highly effective reinforcement learning tasks as it does not require supervision. [2] NEAT attempts to gain advantage when compared to any fixed topologies method by evolving neural network topologies along with its weights on any benchmarking reinforcement learning tasks. The increase in efficiency is due to: [2]

1. Employing a principled method of crossover of different topologies
2. Use of speciation to protect structure
3. Incremental growth from minimal structures

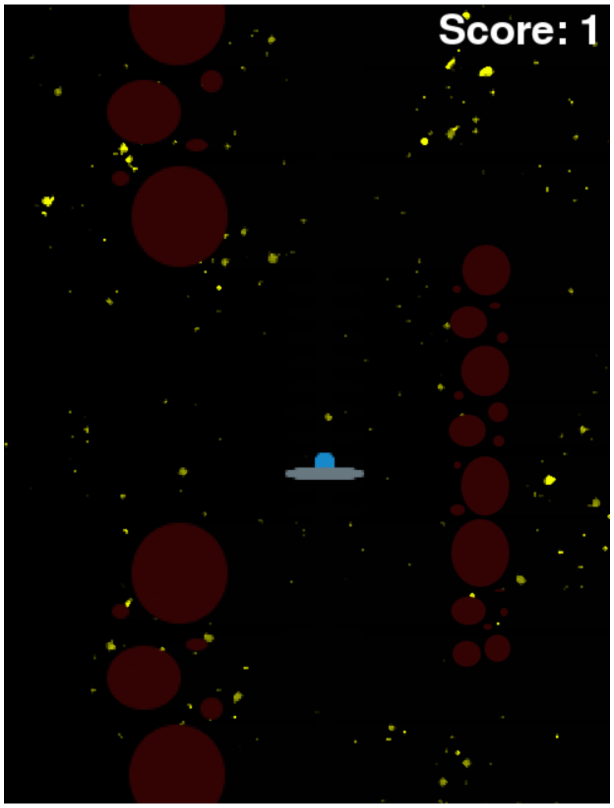
Furthermore, neural networks are a good class of decision making systems to evolve because they are efficient in mapping from genotype to phenotype and also capable of representing solutions to different kinds of problems.

In the current implementation of NEAT for Python, a population of individual genomes with two sets of genes is maintained to build an artificial neural network. [1] Node genes with specifying a single neuron and connection genes, each of which specifies a single connection between neurons. To evolve a solution to a problem, the fitness function computes a single real number indicating the quality of an individual genome. A higher score indicates a better ability to solve the problem and with each reproduction and mutation operation after a generation may add nodes or connections to the genomes. Thus, as the algorithm proceeds, the complexity within genomes may become more complex. When one individual exceeds the user specified fitness threshold or when the pre-set number of generations is reached, the algorithm terminates.

Game

For the purposes of creating a solution to an infinite side scrolling game, an Unidentified Flying Object (UFO) game was created using the PyGame engine, the objective of the game is to avoid and pass as many obstacles as possible. To create a game environment in Python, PyGame was chosen. The PyGame module is widely used to create games within Python. It provides foundational mechanics to create games such as: ability to render sprites, ability to manage and detect time in-game and, the ability to input actions that the sprites can perform. In this situation, rather than having a user input the actions that the UFO can take, the NEAT algorithm was used in order to make the movement decisions for the UFO.

The game that was created was inspired by both the Helicopter game and by Flappy Bird. In this game, there are UFOs that are trying to navigate themselves through an asteroid field. The purpose of this game is to survive and pass as many asteroid obstacles as possible. The UFO loses once it has either collided with an asteroid obstacle or if it flies off screen.



In this game, there are two types of asteroid obstacles. The first is an asteroid belt that spans the height of the screen and contains a gap that the UFO can pass through. The second is a group of asteroids that spawns in the middle of the screen, here the UFO can either go above or below to avoid the collision. The order of these asteroid objects and their position on the screen are randomized and are different in each run of the game. This forces the player, in this case the NEAT algorithm, to make a determination of what to do based on the changing conditions rather than simply relying on a repeated order of actions that can be taken every run.

To create this game, three components were utilized: the main script, the classes of the objects and the configuration for the NEAT algorithm.

The main script focused on: utilizing PyGame to render the game, adding the functionality of the game, and implementing the NEAT algorithm in order to generate genomes to run through and learn how to play the game. It works by looping through a process of: initializing the run, checking for the existence of UFOs and an obstacle, utilizing the NEAT algorithm in order to make a decision, adjust the fitness based on whether a death scenario occurred or the score increased, and adding obstacles if necessary. This process would go on until the UFOs died and either all the genomes had been run through or a fitness threshold had been reached.

The second component were the classes of the UFO and asteroid objects. The UFO class was initialized with the coordinates of the object and considerations for the time attributes. Various functions were created to explain the scope of the UFO's movements within the game. Jump function increases the vertical velocity to a negative number as the PyGame interface measures coordinates from the upper left corner. Similarly, the function move, stay and fall help the UFO move by either maintaining the same coordinate position of the PyGame screen or having it descent in the downwards direction. The UFO always maintains its x position as the obstacles are randomly generated and moved towards the left direction. The UFO then has to then either maintain its position, jump or descend downwards in the screen to pass from one obstacle to the next. In addition to these functions, the UFO class also had a function to draw the UFO and to get the positioning of the UFO based off of the mask within PyGame.

The obstacle class was initialized with its starting x coordinate and had a defined velocity at which it moved across the screen. There were two types of obstacles that were present. One was an asteroid belt with a small gap of 200 pixels that would be present in the middle of the window, the second was a collection of asteroids that would generate in the middle of the window forcing the UFO to dodge them. The order of these obstacles for each run is random. This was designed by having the obstacle type be randomly chosen when the obstacle object is initialized. Like the UFO class, the Obstacle class also had functions to define its movement and to draw the obstacles in the window. One of the most important components of the obstacle class was a function for collision detection. Within the class, a function had been written to obtain the mask of the UFO and to check if the pixels of the UFO overlapped with any of the obstacles that were present on the screen. If there was overlap between the masks of the UFO candidate and the obstacle, the UFO candidate would die.

The third component is the NEAT algorithm configuration. This is a text file that contains the hyperparameters and the values for the hyperparameters for the neural network that was used. The hyperparameters include factors such as: the activation functions for the neurons, the fitness criterion, the mutation rate, the addition or subtraction of neurons, the number of inputs and outputs, just to name a few.

NEAT Hyperparameters and Tuning

The fitness_criterion function was used to compute the termination criterion from the set of genome fitnesses. "max" value was used as the UFO game is an infinite side scroller where a finely tuned algorithm should potentially be able to go through an infinite number of obstacles. To ensure that the fitness_criterion has a threshold value to terminate the evolution process, a fitness_threshold value of 1000 was assigned as passing through a 1000 obstacles would be a pretty robust indicator of the fitness of the candidate. The population size was set to 10 candidates per generation. If the candidates remain stagnated, i.e. not die and not improve fitness scores, a new population group would be created. However, the UFO game will constantly send player-killing obstacles, and so such a condition was set to "false" as it was not needed.

The [Default Stagnation] section that specifies the parameters for the built in "DefaultStagnation" class is only necessary for stagnation implementation. The max_stagnation value of 20 denotes the number of generations that will be considered stagnant and removed if the species do not show improvement. The spiecies_elitism parameter was selected as 2 to prevent total extinctions caused by all species becoming stagnant before new species arise. A setting of 2 will prevent 2 species with the highest fitness from being removed for stagnation regardless of the amount of time they have not shown improvement. Similarly, Reproduction parameters were implemented using the

[DefaultReproduction] section with the elitism value of 2 to determine at least 2 of the fittest members in each species that will be preserved as is from one generation to the next. The survival_threshold marking the fraction for each species allowed to reproduce each generation to default to 0.2.

The [DefaultGenome] was specified in the config file, with activation_default parameter assigned to the new nodes as the tanh function. This is because the tanh function is nonlinear in nature, allowing for stacking. It is bound to the range (-1,1) and has a gradient stronger than sigmoid function resulting in easier optimization despite the vanishing gradient problem. The probability of mutation replacing the node's activation function with a randomly-determined member of activation options was set to 0. The node aggregation-options is a space separated list of the aggregation functions used for nodes. This defaults to "sum" for new nodes. The bias attribute for new nodes was set to a mean value of 0.0 for the new nodes. The bias value for the value of the new nodes was selected using the standard deviation of 1.0 where the bias max and min value was set to +30 to -30. The bias mutation properties such as bias_mutate_powe that marks the standard deviation of the zero centered normal/gaussian distribution from which a bias value mutation is drawn was set to 0.5. Similarly, the bias_mutate_rate which defines the probability that a mutation will change the bias of a node by adding a random value was set to 0.7 to have high variability between candidates of each generation. To avoid the changes caused by a mutation to be too big, the bias_replace_rate probability of a mutation replacing the bias of the node with a new random value was lowered to only 0.1.

The compatibility_disjoint_coefficient was given a value of 1 to linearize the disjoint and excess gene counts' distribution to the genomic distance. This was coupled with the 0.5 as the coefficient for each weight, bias or response multiplier's contribution to the genomic distance, especially for each of the homologous nodes/connections. Furthermore, this coefficient is also used as the value to add for differences in activation functions, aggregation functions, or enabled/disabled status. [3] The probability that a mutation will add a connection between existing nodes and the probability that a mutation will delete an existing connection was set to 50% to allow for neutral connection variability between existing nodes. As a counter measure of a mutation disabling a connection, the value of enabled_mutate_rate probability that a mutation will replace the enabled status of a connection to only 1%. The probability that a mutation will add a new node by replacing an existing connection and similarly the probability that a mutation will delete an existing node and all connections to it was set to 20%. The network parameters were adjusted by adding 0 hidden nodes to be added to each genome in the initial population. 3 input nodes denoting the distance from the top of the pipe opening, bottom of the pipe opening and the distance to the next asteroid column. this would output 1 output node value of the jump function.

The response multiplier attribute for new nodes was assigned a gaussian distribution with a mean value of 1.0, a standard deviation of 0.0, and the minimum and maximum allowed response by the multiplier was so to -30 and +30 respectively. To avoid the chances of a mutation changing the response multiplier of a node by adding a random value, the response_mutate_rate probability was set to 0. Similarly the gaussian distribution for the weight attributes for new connections had a mean value of 0.0, standard deviation of 1.0 with min/max value ranging between -30/+30. The weight_mutate_power that denotes the standard deviation of the zero-centered distribution from which a weight value is drawn is set to 0.5. The probability for a mutation changing the weight of a connection by adding a random value was kept high at 80% for variability. As a countermeasure, the weight_replace_rate probability was set to 10% to decrease the probability that a mutation will replace the weight of a connection with a newly chosen random value.

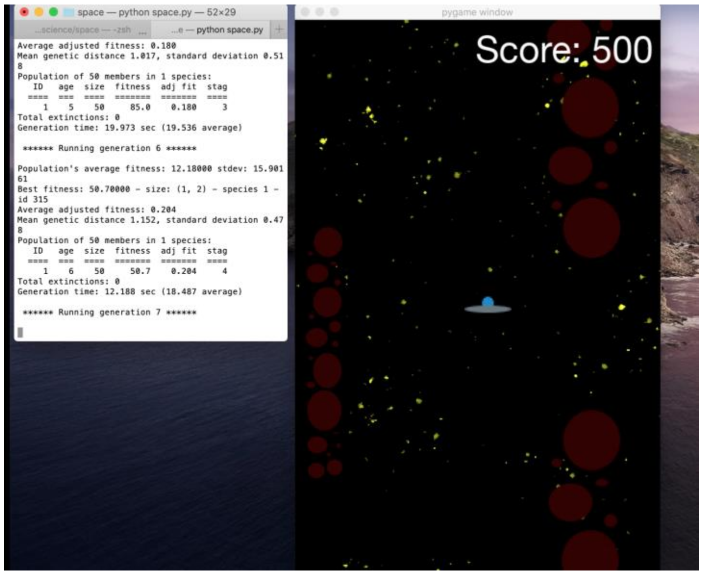
Example Run

The algorithm performance varied every time it was run, this is due to the random nature of the course. Below is the generation output statistics for a demo run that was performed.

```
****** Running generation 0 ******

Population's average fitness: 3.32000 stdev: 4.98422
Best fitness: 38.10000 - size: (1, 3) - species 1 - id 11
Average adjusted fitness: 0.028
Mean genetic distance 1.271, standard deviation 0.433
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    0    50     38.1     0.028     0
Total extinctions: 0
Generation time: 9.124 sec

****** Running generation 1 ******

Population's average fitness: 8.40200 stdev: 15.16326
Best fitness: 97.40000 - size: (1, 3) - species 1 - id 69
Average adjusted fitness: 0.064
Mean genetic distance 1.239, standard deviation 0.533
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    1    50     97.4     0.064     0
Total extinctions: 0
Generation time: 22.555 sec (15.840 average)

****** Running generation 2 ******

Population's average fitness: 13.17200 stdev: 29.88651
Best fitness: 178.00000 - size: (1, 3) - species 1 - id 69
Average adjusted fitness: 0.062
Mean genetic distance 1.190, standard deviation 0.464
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    2    50    178.0     0.062     0
Total extinctions: 0
Generation time: 35.601 sec (22.427 average)

****** Running generation 3 ******

Population's average fitness: 26.45600 stdev: 24.30668
Best fitness: 108.10000 - size: (1, 3) - species 1 - id 69
Average adjusted fitness: 0.228
Mean genetic distance 1.072, standard deviation 0.497
Population of 50 members in 1 species:
```

```
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    3    50    108.1    0.228     1
Total extinctions: 0
Generation time: 23.230 sec (22.627 average)

****** Running generation 4 ******

Population's average fitness: 12.23000 stdev: 11.51497
Best fitness: 27.40000 - size: (1, 3) - species 1 - id 69
Average adjusted fitness: 0.396
Mean genetic distance 1.076, standard deviation 0.468
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    4    50     27.4    0.396     2
Total extinctions: 0
Generation time: 6.735 sec (19.449 average)

****** Running generation 5 ******

Population's average fitness: 17.18800 stdev: 24.02826
Best fitness: 85.00000 - size: (1, 3) - species 1 - id 289
Average adjusted fitness: 0.180
Mean genetic distance 1.017, standard deviation 0.518
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    5    50     85.0    0.180     3
Total extinctions: 0
Generation time: 19.973 sec (19.536 average)

****** Running generation 6 ******

Population's average fitness: 12.18000 stdev: 15.90161
Best fitness: 50.70000 - size: (1, 2) - species 1 - id 315
Average adjusted fitness: 0.204
Mean genetic distance 1.152, standard deviation 0.478
Population of 50 members in 1 species:
ID   age  size  fitness  adj fit  stag
====  ===  ====  =======  =======  ====
1    6    50     50.7    0.204     4
Total extinctions: 0
Generation time: 12.188 sec (18.487 average)

****** Running generation 7 ******
```

Here the program was manually exited as it was optimized and had achieved a score greater than 500.



```
Average adjusted fitness: 0.180
Mean genetic distance 1.017, standard deviation 0.51
8
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    5    50     85.0    0.180     3
Total extinctions: 0
Generation time: 19.973 sec (19.536 average)

 ****** Running generation 6 ******

Population's average fitness: 12.18000 stdev: 15.901
61
Best fitness: 50.70000 - size: (1, 2) - species 1 -
id 315
Average adjusted fitness: 0.204
Mean genetic distance 1.152, standard deviation 0.47
8
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    6    50     50.7    0.204     4
Total extinctions: 0
Generation time: 12.188 sec (18.487 average)

 ****** Running generation 7 ******
```

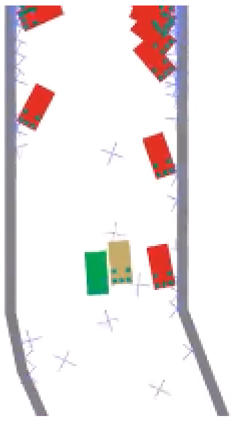
It can be seen that the fitness of the algorithm improves over time. In this run, by the seventh generation, the algorithm had found success and learned how to play the game. Anecdotally, this was one of the more successful runs. Due to the random nature of the course and based on the hyperparameters that were used, there were times where the algorithm would face challenges in learning certain situations. A prime example of this is the algorithm learning how to move downwards. One of the trends that was noticed is that the algorithm preferred the UFO to move towards the top and would rather move upwards than descend down. Over several iterations, it can be seen that the algorithm when detecting an obstacle in the middle of the screen, would move upward to avoid it, it then would generalize this behaviour to when there was also an obstacle towards the top of the screen, resulting in it flying off screen and dying. This was an area that proved to be a challenge for the algorithm as it did not demonstrate a strong capability of learning the boundaries where it would die, it mainly learnt that if it collides with an obstacle that it would die. This issue could be solved in a number of ways. The first method would be to give the algorithm a larger number of generations to learn from. For the purposes of this investigation, there was a preference for limiting the number of UFOs and limiting the number of generations, however, if left to run longer, it is likely that the algorithm would have eventually detected this behaviour. Another area for improvement is with the actual neural network itself. There are a number of hyperparameters that could be tested on to find the optimal performance to solve this game. Lastly, one method that could be used to address this behavior would be to establish the boundaries of the game more clearly. When the UFO is near the top of the screen, it is unintuitive for it to fly off the screen, the optimal decision at that point would be to try to dodge the obstacle in the other direction. Adjusting the mechanics of the game could help to solve this issue and would be valuable to explore in the future.

Challenges

Throughout the development of the game, there were a number of challenges that were faced. One of these challenges was building the mechanics of the game, and the other was the emergence of guaranteed death scenarios.

Building the mechanics of the game proved to be challenging at times. Where PyGame provides quite a bit of functionality, especially when it comes to drawing and working with the sprites, one of the challenges was working with the time. When running the game, your sprites have to consider the time delta when moving. An issue that was run into, was the sprite for the UFO initially would not move smoothly and would vibrate across the screen. This was a problem because it was difficult to look at and it would vibrate when passing the obstacles increasing the likelihood of collision.

Another challenge that was faced was that there are instances where the UFO faces a guaranteed death scenario. This was first discovered in the development process. Initially, the concept was to have a terrain that the UFO would have to navigate through, similar to the example of deep learning with cars. [4] One of the issues that presented here was developing the terrain such that there would be enough room for the UFO to navigate, Often due to the random nature of the obstacles, there would be instances where the UFO had no way to get through. This poses a problem when it comes to the neural network learning how to play the game. After moving away from the terrain idea to the asteroid belt, another death type situation occurred where it was identified that the UFOs were not able to move fast enough to avoid the object, despite going in the right direction. These death situations pose a problem in having the neural network learn from the generation because with a small set of genomes, it becomes a form of nonsense data. The neural network is getting a set of inputs and performs actions that should result in a positive outcome. Getting the opposite result, and having a small set of genomes with these death scenarios, where the UFO object does not have a control over, can result in it learning that the right course of action is incorrect.



Conclusion:

Overall for this project, a infinite side scrolling game was created and the NEAT algorithm was applied in order to train a neural network to learn how to play the game. The game that was created was inspired by the helicopter game and was of a UFO trying to avoid asteroid objects. The NEAT algorithm learned to play the game well when given an appropriate amount of time to train. When playing the game, it was discovered that the stated mechanics of the game had an impact on the actions that the neural network occurred. It was also discovered that the neural network learned certain strategies, for example, staying at the higher end of the board. With hyperparameter tuning, the algorithm can learn how to play the game with a higher fitness level, making AI a useful tool to play these types of side scrolling games.

Sources

[1] "NEAT Overview." *NEAT*, neat-python.readthedocs.io/en/latest/neat_overview.html.

[2] Kenneth O. Stanley and Risto Miikkulainen, 2002, Efficient Evolution of Neural Network Topologies , Department of Computer Sciences The University of Texas at Austin Austin, TX 78712, Retrieved fromF <http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf>

[3] "Configuration File Description." *NEAT*, <https://neat-python.readthedocs.io/en/latest/config_file.html>

[4] Samuel, Arzt, Deep Learning Car, Retrieved from< https://www.youtube.com/watch?v=Aut32pR5PQA>

[5] Tech With Tim, AI Teaches Itself to Play Flappy Bird - Using Neat Python! Retrieved From <https://www.youtube.com/watch?v=OGHA-elMrxI>